Specialized Hardware Supplement

to the

Software Communication Architecture (SCA) Specification

JTRS-5000 SP
V3.0
27 August 2004

Prepared by the
Joint Tactical Radio System (JTRS) Joint Program Office

Contributions made through
Workshop on SCA Extensions
To support Signal Processing Subsystem
29-30 April 2004

Revision Summary

3.0	Formal release for initial validation.

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

The Specialized Hardware Supplement (SHS) to the JTRS Software Communication Architecture (SCA) Specification specifies how to improve portability of software for processing elements other than general purpose processors. This supplement specifies:

- A Hardware Abstraction Layer Connectivity (HAL-C) specification,
- A reduced POSIX AEP for DSP environments,
- Waveform functional blocks to be provided as part of each radio set, and
- An Application Interface for antenna interfaces.

1.1 SCOPE OF SPECIALIZED HARDWARE SUPPLEMENT

This document supplements the SCA Specification by specifically addressing the software for specialized hardware (Field Programmable Gate Arrays (FPGA) and Digital Signal Processors (DSP) and Application Specific Integrated Circuits (ASIC)). The requirements in this supplement are intended to reduce the cost of portability by mitigating a set of problems for this type of software. These problems include the lack of standard operating systems in DSPs and the differing computational paradigms of DSPs, FPGAs, ASICs, network processors and other special function devices.

This supplement applies to non-CORBA components executing on Specialized Hardware. It does not change any requirements in the main Software Communications Architecture (SCA) specification and relies on non-CORBA components being deployed in accordance with the SCA.

While this supplement specifically addresses DSPs, FPGAs, and ASICs, it is intended to be generic enough to apply to emerging hardware such as computing grids, network processors, special function devices, and partially reconfigurable FPGAs.

All requirements are indicated by the word “shall”. If no “shall” appears in a sentence, it is not a requirement.

1.2 REFERENCE DOCUMENTS

The Open Group, ISO/IEC 9899:1990, *"Programming languages - C"*

Object Management Group, final dtc/04-05-04, *“PIM and PSM for Software Radio Components Final Adopted Specification”*

The Open Group, ISO/IEC 9945-1:1996 Information technology – *“Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) [C Language]”* {includes 1, 1b and 1c}

IEEE Std 1003.5c-1996
IEEE Standard for Information Technology, *“Information Technology - POSIX - Ada Language Interfaces - Part 1: Binding for System Application Program Interface - Amendment 1: Realtime Extensions”*

2 HAL CONNECTIVITY (HAL-C)

The Hardware Abstraction Layer Connectivity (HAL-C) specifies a hardware platform-independent means for communication between software components running on specialized hardware.

The amount of waveform-specific software running on special purpose hardware is increasing as JTRS grows to cover high-rate waveforms such as those operating above 2 GHz. Currently, the software components running on special-purpose hardware use JTR Set-specific mechanisms to communicate. The choice of communication mechanism fundamentally affects the software design. For example:

a) The API used to exchange control and data messages determines whether the software component is always running or whether it performs some computation and then halts until a new message arrives.

b) The choice of connection-oriented vs. connectionless communications determines the overall system structure. Connection-oriented mechanisms require fixed data flows, link initialization, and unique addresses. Connectionless communication mechanisms allow more random communication patterns.

c) Use of different APIs for different physical transports between software components allows and encourages specialization of the software design to the layout of the components on a particular hardware platform.

The HAL-C specifies a communications API and thereby minimizes the effect of the hardware platform's communication mechanisms on the software design. This reduces the probability of significant component rewrites during porting.

Overall lifecycle costs are also reduced by specifying a high-level abstraction API that isolates the waveform-specific software from the radio set-specific software.

2.1 DEFINITIONS

A *processing element (PE)* is a hardware component capable of performing computational functions and making decissions based on prior state. A general-purpose processor is a PE, as are a DSP, FPGA, ASIC, or other specialized hardware devices. In this document, the term *CORBA-enabled PE* will be used whenever referring to any PE that offers a CORBA implementation and is therefore capable of supporting SCA operations. The term *PE* by itself refers to a hardware component that does not support CORBA and cannot issue or implement SCA API calls directly.

A *HAL-C Component (HC)* is a component that performs processing functions as part of a waveform implementation. Many HCs may run on a given PE. Normally HCs are software written by the waveform developer and hosted on the target radio set. However, in some cases HCs may be software or hardware that is provided by the hardware platform developer. An example is a hardware accelerator for a turbo decoder. In this document, the term *HC* by itself refers to a functional component that cannot issue or receive CORBA calls directly because it is hosted on a PE that does not support CORBA.

The *HAL-C Infrastructure* is software that realizes the HAL-C APIs and enables the transfer of data between endpoints.

In this supplement, the term *platform* has several meanings depending upon context. *Hardware platform* is used to refer to the set of PEs on which software will execute. *Radio set platform* is used when the context indicates that the *Hardware platform* is a radio (e. g., in reference to an antenna or waveform). Unmodified p*latform* is used to refer to an object in which the radio set is to be imbedded (e. g., an airplane or satellite)

2.2 SCOPE OF HAL-C

HAL-C specifies the following:

a) A flexible, scalable abstraction that can be implemented on different hardware architectures and hardware platforms.

b) A method and API by which HCs can efficiently interface with HCs on other PEs (location-transparent) via high-speed transport mechanisms provided by the hardware platform. Note, because HAL-C is location-transparent it also supports interfacing two HCs on the same PE.

c) A method for the connectivity between HCs to be specified externally rather than compiled into the HCs. (For example, by a SAD file in the current SCA).

HAL-C has no effect on the SCA requirements that apply to components running on CORBA-enabled PEs. HAL-C only applies to HCs loaded on special purpose hardware.

HAL-C does not currently specify the mechanisms used to establish the connectivity between HCs based on an external specification such as the SCA SAD file. Since Both the core framework and the HAL-C Infrastructure are specific to the radio set, there is no need to restrict the manner in which the core framework configures the HAL-C Infrastructure. If desired, common initialization mechanism could be specified by future extensions to the HAL-C.

HAL-C is designed to enable redirection of flows by sophisticated implementations. This allows JTR sets to exploit this capability to dynamically allocate resources on waveform startup from a pool of available PEs as well.

Redirection of flows and reallocation may also be useful for fault tolerance and Quality of Service features. The current HAL-C specification does not support initiation of these operations by waveforms, but rather by the hardware platform. HAL-C has been designed to allow for extension to support features such as channel unbinding and binding and flow stopping and starting, which will be needed to standardize the way a waveform reallocates its resources.

2.3 HAL-C MODEL

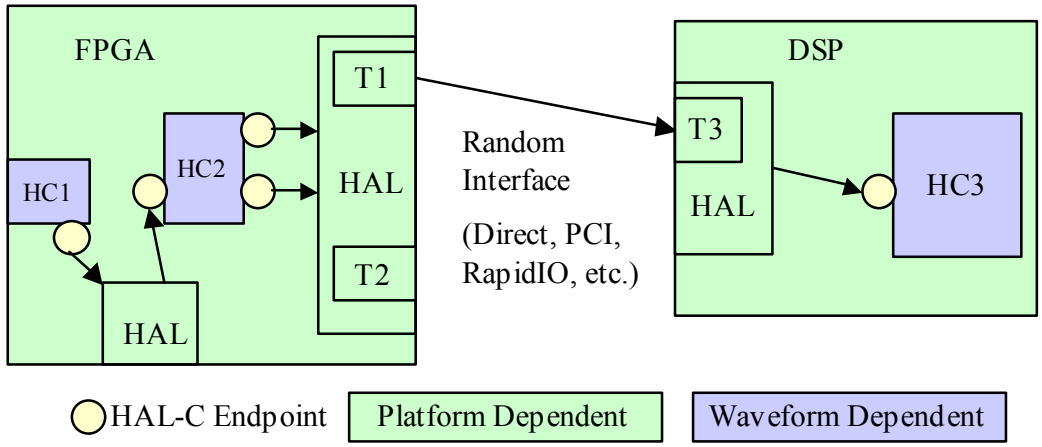


Figure 1: HAL-C Model

Figure 1 shows the implementation model of HAL-C. A HAL-C realization consists of both Waveform and JTR set-specific components.

A HAL-C component (HC) implements application functionality. These components are written by the waveform developer and designed to the HAL-C API so that their portability is maximized.

The JTRS Radio Sets shall implement the corresponding HAL-C API for each non-CORBA processing element (PE). On FPGAs and other similar non-processor-based PEs, a subset of the HAL-C functionality is available and is specified in section 2.4.2.

The API realization implemented by the platform provides an infrastructure within which the HCs will operate. The main purpose of this infrastructure is to enable communication between HCs that reside on the same and/or different PEs.

While this specification does not detail a transport abstraction implementation, it is useful to describe it as a single transport module that resides in the HAL-C infrastructure, represented as Tx. There are two types of transports to describe. The first is the Physical Channel that connects two different PEs and facilitates the sending of data and commands between two HCs residing on different PEs. The Physical Channel is considered the mapping of the transport onto the hardware platform. The other type is an internal transport implementation that is hardware platform-specific but is required to align with the interface of the HAL-C APIs.

To simplify the connection aspect of the model, the notion of Endpoints is introduced. The path out of a HC and through a transport module will be referred to as the Logical Channel.

An application uses the HAL-C API to access Endpoints, and to send and receive data; the HAL-C infrastructure moves data to the specified Endpoints across the logical channel.

2.4 HAL-C API

The HAL-C specification details how two HCs exchange data. This specification only prescribes the transfer of raw data bits. No meaning is ascribed to the raw data. For PEs that support a C runtime, the HAL-C API consists of 4 functions that implement retrieving Endpoint handles, sending data to Endpoints, receiving data at endpoints, and registering callbacks. For PEs such as FPGAs, the API consists of a data bus, as well as associated clock and control signals. The HAL-C infrastructure and related APIs are part of the operating environment.

2.4.1 HAL-C for Processing Elements supporting a C runtime

2.4.1.1 Description.

This section details the requirement that a non-CORBA-enabled PE with a C runtime provide the realization of the HAL-C C API. All the functions that are part of the API are described as well as the intended behavior of the infrastructure to support these functions.

2.4.1.2 UML.

N/A

2.4.1.3 Types.

2.4.1.3.1 HalcEndpointHandle.

The first argument of each API function is a handle. This handle is an opaque type that identifies the intended endpoint to the infrastructure implementation. It may, for example, be an integer that indexes into a table of hardware platform-specific data structures containing buffer space and information about how to route data to other endpoints.

Endpoints are addressable sink (entry) and source (exit) interfaces of HC components. These endpoints are implemented as part of the HAL-C infrastructure. Associated with each endpoint will be data that differentiates its dataflow from other endpoints sharing the same physical channel. The HAL-C infrastructure will be configured by an outside agent, which will provide all endpoint information. The required information is the name by which a component is referenced and the PE on which it is loaded.

The association of a HC to an endpoint will be through a handle returned by halcGetEndpoint. This allows the HAL-C infrastructure to change the endpoint information dynamically in an effort to reallocate the data flows in the system. Therefore, each HC will be required to supply an initialized handle to the HAL-C API functions it calls.

2.4.1.4 Attributes.

N/A.

2.4.1.5 Operations.

The operations of the HAL-C API will be made available to HC through a library.

2.4.1.5.1 halcGetEndpoint.

2.4.1.5.1.1 Brief Rationale.

HalcGetEndpoint will allow a HAL-C component to get a handle to the named endpoint. This handle will then be used as an input by the other operations of the HAL-C API.

2.4.1.5.1.2 Synopsis.

```
halcResult   HalcGetEndpoint(halcEndpointHandle* eph, char*
componentName, char*  endpointName);
```

2.4.1.5.1.3 Behavior.

The parameters of the halcGetEndpoint operation are:

- eph is a handle to an Endpoint associated with the supplied name.
- componentName is the character string name of component
- endpointName is the character string name of endpoint.

The halcGetEndpoint operation shall match the componentName and endpointName parameters against the configuration tables in the HAL-C infrastructure to identify the desired endpoint.

The halcGetEndpoint operation shall place the identified endpoint handle in the location pointed to by eph.

When later provided in a HAL-C operation, this handle is used by the infrastructure to access the associated endpoint.

2.4.1.5.1.4 Returns.

The halcGetEndpoint operation shall return halcResult with a value of halcSUCCESS to indicate that an endpoint was obtained.

The halcGetEndpoint operation shall return halcResult with a value of halcFAIL to indicate that the endpoint was not obtained.

2.4.1.5.1.5 Exceptions/Errors.

This operation does not return any exceptions.

2.4.1.5.2 halcSend.

2.4.1.5.2.1 Brief Rationale.

HalcSend is a connection-oriented blocking send function and is the method by which the components transfer data.

2.4.1.5.2.2 Synopsis.

```
halcResult   halcSend(halcEndpointHandle eph, uint32 maxLength, uint32*
bytesSent, void* data, uint32 timeout);
```

2.4.1.5.2.3 Behavior.

The halcSend operation shall deliver up to maxLength bytes from the address specified by data parameter to the HAL-C Infrastructure for transfer to the endpoint designated by eph.

- eph is the handle of the endpoint to which the HAL-C Infrastructure delivers the data.
- maxLength is the size of the data buffer.
- bytesSent is the amount of data actually sent where it's value may be < maxLength in the event of a timeout.
- data is the buffer where data is stored. The caller retains ownership of the data.
- timeout is the amount of time in microseconds ((2**32-1) = forever is default) that has been allotted for successful completion of the operation.

The invoking application will receive control as soon as data has been delivered to the HAL-C Infrastructure. Transfer of data to the endpoint will continue asynchronously.

2.4.1.5.2.4 Returns.

The halcSend operation shall return halcResult with a value of halcSUCCESS to indicate that data has been delivered for transmission.

The halcSend operation shall return halcResult with a value of halcTIMEOUT to indicate data had not been delivered after timeout microseconds.

2.4.1.5.2.5 Exceptions/Errors.

This operation does not return any exceptions.

2.4.1.5.3 halcReceive.

2.4.1.5.3.1 Brief Rationale.

HalcReceive is a connection-oriented blocking receive function.

2.4.1.5.3.2 Synopsis.

```
halcResult  halcReceive(halcEndpointHandle eph, uint32 maxLength,
uint32*   bytesReceived, void* data, uint32 timeout);
```

2.4.1.5.3.3 Behavior.

The halcReceive operation shall block until data is available or a timeout occurs.

The halcReceive operation shall transfer up to maxLength bytes of data to the address specified by the input parameter, data.

The halcReceive operation shall return the output parameter, bytesReceived, with a value indicating the number of bytes transferred.

- eph is the handle of the endpoint from which to receive data.
- maxLength is the size of the data buffer.
- bytesRecieved is the amount of data actually received.
- data is the buffer where data is stored. The caller retains ownership.
- timeout is the amount of time in microseconds ((2**32-1) = forever is default) that has been allotted for successful completion of the operation.

The halcReceive operation shall not transfer data from its buffer if the length of data available for transfer is greater than maxLength,.

2.4.1.5.3.4 Returns.

The halcReceive operation shall return halcResult with a value of halcOVERFLOW to indicate that the data to be transferred is longer than maxlength.

The halcReceive operation shall return halcResult with a value of halcTIMEOUT if the operation is not completed within the time specified by the input parameter timeout.

The halcReceive operation shall not return halcResult with a value of halcTIMEOUT if the input parameter timeout has a value of 0.

2.4.1.5.3.5 Exceptions/Errors.

This operation does not return any exceptions.

2.4.1.5.4 halcRegisterCallback.

2.4.1.5.4.1 Brief Rationale.

The callback function will provide the facility for the infrastructure to send data to a component without the component blocking on a receive call. The data will arrive from a specific endpoint or get generated by some other means within the implementation of the infrastructure, i.e., events, errors, etc.

2.4.1.5.4.2 Synopsis.

```
typedef void (*halcCallback)(void* token, halcResult status, uint32
length, void* data);

halcResult  halcRegisterCallback(halcEndpointHandle eph, halcCallback
cb, void* token, uint32 timeout);
```

2.4.1.5.4.3 Behavior.

The halcRegisterCallback operation shall register the function designated by the input parameter halcCallback with the Endpoint designated by eph.

If input parameter halcCallback is null, the halcRegisterCallback operation shall deregister any callback associated with the endpoint designated by eph.

- eph is the handle of the endpoint on which to perform the operation.
- cb is the callback function pointer.
- token is a label that enables the callback function to locate relevant data structures.
- timeout is the amount of time in microseconds ((2**32-1) = forever is default) that has been allotted for successful completion of the operation.

2.4.1.5.4.4 Returns.

The halcRegisterCallback operation shall return halcResult with a value of halcSUCCESS to indicate that the halcCallback has been registered.

The halcRegisterCallback operation shall return halcResult with a value of halcFAIL to indicate that the halcCallback has not been registered.

2.4.1.5.4.5 Exceptions/Errors.

This operation does not return any exceptions.

2.4.1.6 HAL-C Infrastructure Behavior

The HAL-C Infrastructure provides the hardware platform-specific realization of the HAL-C APIs and performs the data transfer to endpoints. The HAL-C Infrastructure moves data it receives to the specified endpoint and performs the appropriate actions when callbacks have been registered.

Upon receipt of data from the halcSend operation, the HAL-C Infrastructure shall transfer the data to the specified endpoint.

The HAL-C infrastructure shall invoke the registered Callback function with the parameter halcResult set to halcTIMEOUT if the registered timeout interval has elapsed since the last data arrived.

Upon receipt of data at an endpoint where a callback has been register, the HAL-C infrastructure shall invoke the registered Callback function.

2.4.2 HAL-C for FPGAs

2.4.2.1 Description

The SCA defines two interface types: a “provides” port that provides a service, and a “uses” port, which uses that service. This concept will be extended inside the FPGA to the component level – a component is provided data from the HAL-C infrastructure using its provides port, and “uses” the HAL-C infrastructure to provide data to another component via its uses port (or endpoint). In an FPGA context, we will use the terms source and sink to define the SCA port equivalent within this document.

“Source” interfaces are used to pass data from the HAL-C component to the HAL-C infrastructure, which will route information to the appropriate component based upon hardware platform implementation decisions.

“Sink” interfaces accept data driven to the HAL-C component from the HAL-C infrastructure.

The HAL-C API expects the HAL-C components to provide or accept data over a common interface. On the FPGA this interface takes the form of a specification of a set of wires and signals.

The Source interface shall output the following signals:

- *clock* – All signaling on this interface is synchronous to this clock.
- *data* – This bus is used to carry payload data. Data bus width will be defined later.
- *channel* –This bus defines the logical channel number associated with a data transfer.
- *length* – This bus defines the length in words of the data buffer to be transferred. A size of MAXBUFFERSIZE indicates that data is to be constantly streamed.
- *write* – Asserted to transfer data on the *data* field.

- *socketRequest* – This MAXSOURCESOCKETS-wide vector indicates that a specific HAL-C logical channel interface is requesting access to the sink logical channel.

The Sink interface shall receive the Source output signals.

The Source interface shall transfer a block of data of size indicated by the length signal.

The Sink interface shall receive data, the size of which is indicated by the length signal.

The Sink interface shall output the following signal:

- *socketReady* – This MAXSINKSOCKETS-wide vector contains a flow control signal from each Sink interface accepting data from the Source interface.

The source interface shall receive the *socketReady* signal.

The Source interface is the hardware equivalent of halcSend; The Sink interface is the hardware equivalent of halcReceive. The following figures detail the interface definitions:

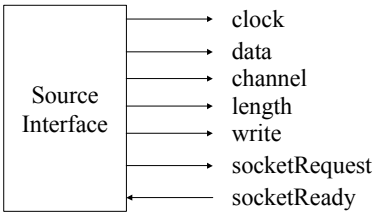


Figure 2: Source Interface Definition

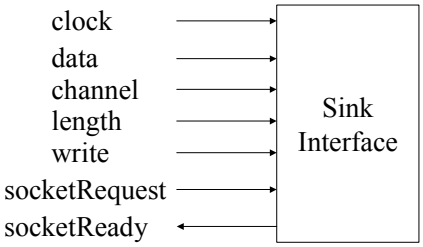


Figure 3: Sink Interface Definition

For the FPGA implementation, there is not a concept of the halcGetEndpoint or halcCallback functions. The initialization of the endpoints will be accomplished through the HAL-C infrastructure. In keeping with the signal definitions, the HAL-C Infrastructure shall supply the HC with the logical channel number (handle) to use.

It is permissible and expected that all unused signals will be removed during optimization.

2.4.2.2 HAL-C Infrastructure Behavior

Because FPGAs do not have an equivalent to the halcGetEndpoint and halcCallback functions, this information is specified as part of preparing the FPGA load. In order for an application to receive the address of an endpoint, an HC component shall reserve register locations that will allow the infrastructure to store the each endpoint's logical channel number as a 32bit integer. This register's address and other access information would then be described as part of the application documentation.

2.4.2.3 FPGA Layout

An FPGA is a programmable hardware device, and as such, functional components within an FPGA are interconnected through direct wire connections or bus architectures. Connections between HAL-C and the Endpoints within waveform components can be made during the generation of the load map or treated the same as external connections. External connections between an FPGA and other PEs will be completed through the HAL-C configuration done at load time.

This means that the internal architecture of an FPGA image containing multiple functional components can be thought of as a number of smaller discrete FPGAs interconnected through external transport mechanisms. This concept is illustrated in the following figure:

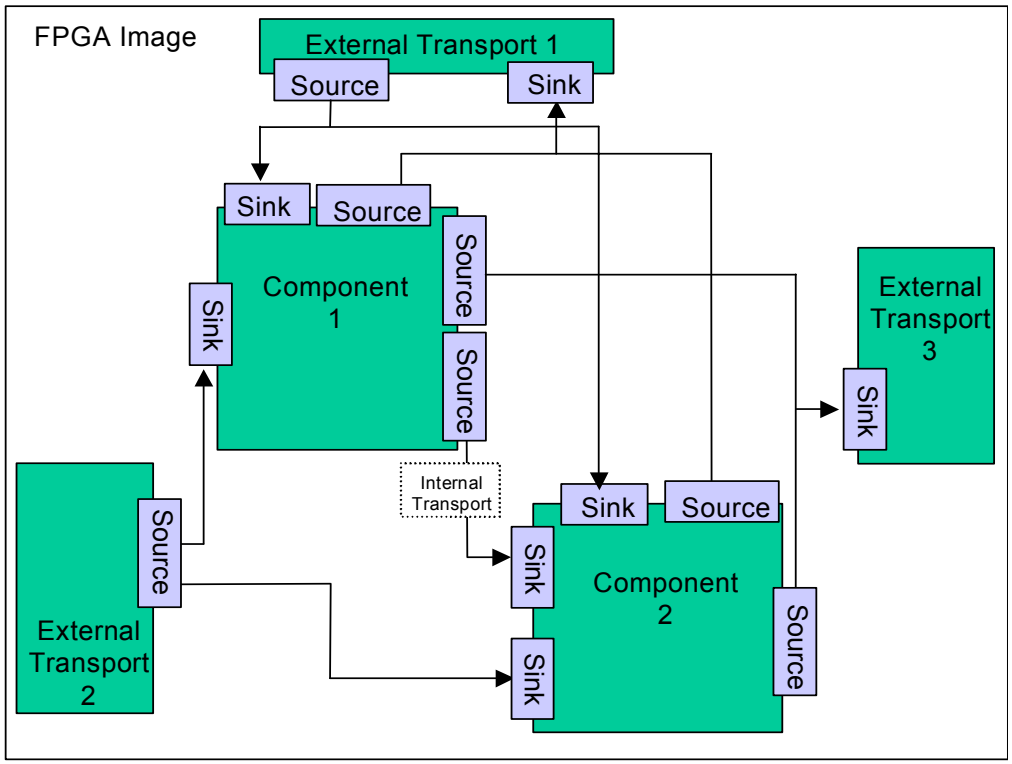


Figure 4: FPGA Layout

2.4.2.4 Flow Control

During a typical data transfer, the Source interface asserts a *socketRequest* signal to gain access to the Sink interface. When the Sink interface is ready, the sink interface asserts a *socketReady* signal. The Source interface then asserts a *write* signal, and on the next clock edge, the first data word, channel number, and buffer length are transferred to the sink interface, and the length is decremented. On subsequent clock edges, the source interface transmits additional data words, and decrements the length until all data words are transferred. The source interface de-asserts the *write* and *socketRequest*, following the transfer of the last data word in the buffer. The sink interface de-asserts the *socketReady* following the transfer of the last data word in the buffer.

The Sink endpoint de-asserts *socketReady* at any time to interrupt the data transfer. This could be done to service a *socketRequest* from a higher priority Source endpoint, or to maintain flow control accounting for discrepancies in the data bus width.

Overall, this scheme assumes that the Sink endpoint consumes data from the Source at a rate greater than or equal to the rate at which the Source is sending data.

For those cases where the socketReady/socketRequest handshaking overhead is unacceptable, data can be constantly streamed, without any handshaking. If the length vector is set to MAXBUFFESIZE then the handshaking shall be disabled. It will be the function of the waveform component to assure that the data is consumed at an appropriate rate.

On interfaces supporting multiple channels per transport, the HAL-C component will invoke the appropriate socketReady/socketRequest handshaking operations.

2.4.2.5 Data bus width

The sink interface shall support the following data bus widths: 1, 8, 16, 32, 64 bits.

Translation between disparate data bus types is accommodated through translation components in the HAL-C infrastructure, which will map the word sizes, pack or unpack data, and change endian-ness, as appropriate. Any necessary translation components will be provided during the porting/integration effort by the platform vendor.

2.4.2.6 HAL-C Components

Figure 5 shows one possible architecture of a HC component. The figure shows the use of multiple endpoints for one waveform component, the separation of control and data, and unidirectional endpoints.

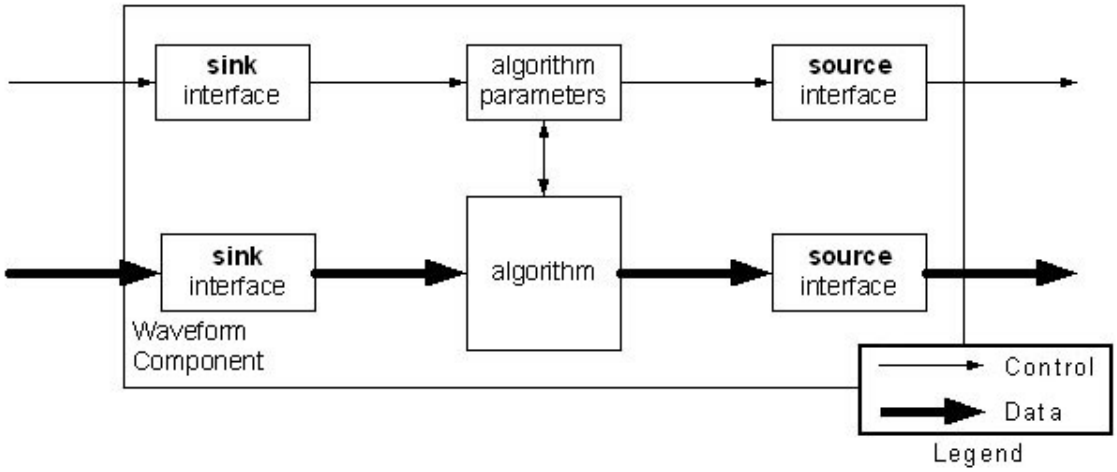


Figure 5: An HC Implementation

3 OS SERVICE APIS FOR DSP ENVIRONMENT

3.1 PURPOSE AND SCOPE

This section defines common Operating System (OS) Services, based upon a POSIX AEP subset extracted from the SCA POSIX AEP. These common services will be implemented across all SCA Digital Signal Processing (DSP) implementations. There is an obvious tradeoff between the robustness of the set of chosen APIs and the complexity required to provide an implementation for those APIs in the DSP environment.

The approach taken to establish a composite list of implementable OS APIs for the DSP environment is:

1. Identify high-priority services based upon their operational benefit.
2. Consider those additional services related to the SCA POSIX AEP profile.
3. Filter for redundancies and for availability in common DSP libraries.

DSP environments vary in supported OS APIs and overall service capabilities. There presently exists no common API set, similar to SCA POSIX AEP subset for GPPs, that establishes a commonly supported interface for all components within the DSP environment. The logical selection of the SCA POSIX AEP APIs as a starting point, and the subsequent mapping of that API set to a baseline set of commonly existing DSP OS APIs, establishes a fundamental set of DSP OS services that are realistic to implement as part of a real-time signal processing environment. The set of APIs initially required has been largely limited to functionality commonly provided as part of many contemporary DSP operating systems. Requiring a larger set of APIs beyond such common functions would likely impose a substantial software development effort (e.g. File Services are not presently part of most DSP OSs, so mandating an implementation of DSP File Services could impose a great burden on hardware platform developers). Also, the features of

HAL-C were considered in the selection of functions that support inter-process communication.

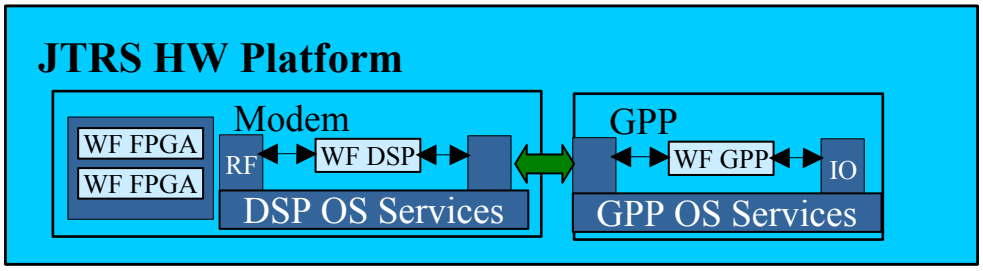


Figure 6: OS Services

3.2 OS SERVICE APIS FOR DSP ENVIRONMENT REQUIREMENTS

The initial review of POSIX APIs identified the following areas as candidates for DSP operating system standardization.

- C-Runtime Library Support
- A simple hardware interrupt structure
- Simple message passing (POSIX.1b- Message Passing)
- Multi-threaded support (POSIX.1c Threads)
- Software timers (POSIX.1b-Timers)

Appendix E: SCA Application Environment Profile for Digital Signal Processors defines the functions and options designated as mandatory. These functions are provided by the DSP. Waveform applications can assume these functions will be provided.

The complete list of functions is specified in Appendix E. The following tables identify the applicable portions of the POSIX standard from which these functions were selected.

Table 1: Required Standards

Standard	DSP AEP
C Standard (ISO/IEC 9899:1990)	PRT
POSIX.1 (ISO/IEC 9945 -1):1996	PRT
POSIX.1b (ISO/IEC 9945 -1):1996	PRT
POSIX.1c (ISO/IEC 9945 -1):1996	PRT
POSIX.5b (IEEE 1003.5 - 1992)	OPT

NOTE:
PRT Partial, only the subset or options or Units of Functionality called out in A.3.
MAN Mandatory, complete with all options.
OPT Optional, may be included in the environment.

Table 2 contains the required options, limits, and any other constraints on POSIX.1b.

Table 2: POSIX.1b Option Requirements

	DSP AEP
{_POSIX_ASYNCHRONOUS_IO}	NRQ
{_POSIX_MAPPED_FILES}	NRQ
{_POSIX_MEMLOCK}	NRQ
{_POSIX_MEMLOCK_RANGE}	NRQ
{_POSIX_MEMORY_PROTECTION}	NRQ
{_POSIX_MESSAGE_PASSING}	NRQ
{_POSIX_PRIORITIZED_IO}	NRQ
{_POSIX_PRIORITY_SCHEDULING}	NRQ
{_POSIX_REALTIME_SIGNALS}	NRQ
{_POSIX_SEMAPHORES}	MAN
{_POSIX_SHARED_MEMORY_OBJECTS}	NRQ
{_POSIX_SYNCHRONIZED_IO}	NRQ
{_POSIX_TIMERS}	MAN
{_POSIX_FSYNC}	NRQ

NOTE:
NRQ Not required for this profile.
MAN Mandatory for this profile.

Table 3 contains the required options, limits, and any other constraints on POSIX.1c.

Table 3: POSIX.1c Option Requirements

Option	DSP AEP
{_POSIX_THREADS}	MAN
{_POSIX_THREAD_ATTR_STACKADDR}	NRQ
{_POSIX_THREAD_ATTR_STACKSIZE}	MAN
{_POSIX_THREAD_PRIO_INHERIT}	NRQ
{_POSIX_THREAD_PRIO_PROTECT}	NRQ
{_POSIX_THREAD_PRIORITY_SCHEDULING}	MAN
{_POSIX_THREAD_PROCESS_SHARED}	NRQ
{_POSIX_THREAD_SAFE_FUNCTIONS}	MAN

NOTE:
NRQ Not required for this profile.
MAN Mandatory for this profile.

The DSP development environment shall provide the functions and options designated as mandatory by the SCA AEP for Digital Signal Processors defined in Appendix E. An OS or services library may provide functions and options in addition to those designated as mandatory by the profile.

Application components (software which executes a subset of total waveform functionality) running on a DSP shall be limited to using the DSP OS services that are designated as mandatory by the SCA AEP for Digital Signal Processors.

HAL-C infrastructures are not restricted to using the services designated as mandatory by the AEP as specified and defined in Appendix E.

4 WAVEFORM FUNCTIONAL BLOCKS

4.1 PURPOSE AND SCOPE

This section identifies Waveform Functional Blocks (WFB) to be made available on all SCA-compliant Special Hardware Components (e.g., DSPs & FPGAs). JTR Set acquisitions will include the functions prescribed by these WFBs. They need not be included as part of a waveform, but will be available in a library for use on each JTR set.

WFBs are not equivalent to SCA or HAL-C components. WFBs are simple functions that may be included in a component much like a library module. Components are aggregations of software units that are deployable and that conform to requirements specified by the component architecture.

JTRS set acquisitions will require that an Executable Implementation Independent Model (IIM) (see the Acquisition Guidance to the JTRS Software Communication Architecture (SCA) Specification section 3.2.2) be generated for each WFB. An Executable Specification is an executable model in a higher order language. The test vectors and test results will be made available with the Executable Specification to validate correct implementation of the WFB across hardware platforms.

4.2 STANDARD MODULATION BLOCKS

The following Waveform Functional Blocks (for modulation) will be made available on SCA-Compliant Special Hardware Components (e.g., DSPs & FPGAs):

- **CCSK** Cyclic Code Shift Keying.
- CPM Continuous Phase Modulation

Waveform Applications need not provide these functions as part of the Waveform Application.

4.2.1 Cyclic Code Shift Keying

Figure 7 describes the interfaces for Cyclic Code Shift Keying. Implementation of these interfaces will be realized differently depending upon whether the processing element is a DSP or FPGA.

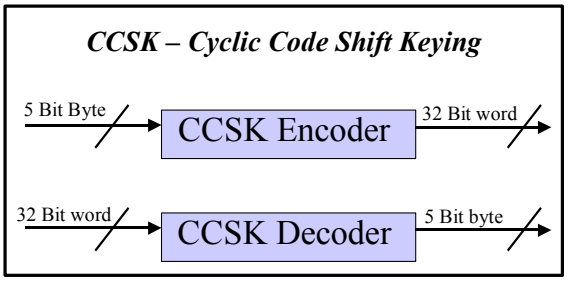


Figure 7: Cyclic Code Shift Keying

4.2.2 Continuous Phase Modulation

Figure **8** describes the interfaces for Continuous Phase Modulation. Implementation of these interfaces will be realized differently depending upon whether the processing element is a DSP or FPGA.

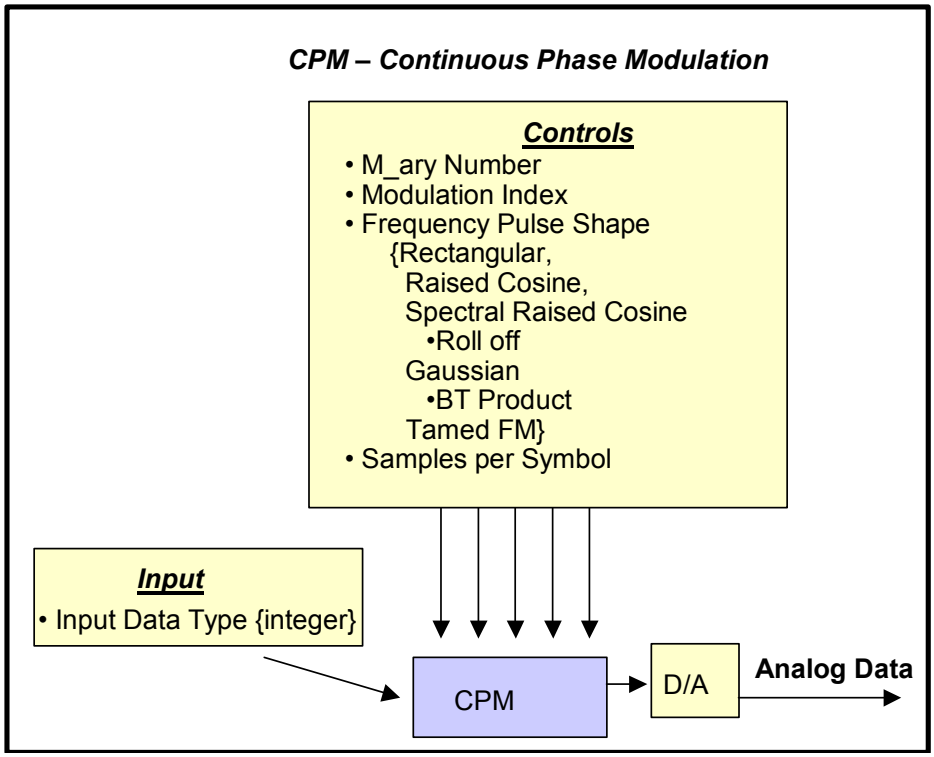


Figure 8: Cyclic Continuous Phase Modulation

4.3 Standard Coding Blocks

Forward Error Correction (FEC) coding blocks can be used to improve the noise immunity. Turbo Code and Reed-Solomon block codes may also include interleaving to improve burst noise immunity.

The following standard Waveform Functional Blocks will be made available on SCA-Compliant Special Hardware Components (e.g., DSPs & FPGAs):

- **Turbo Encoder/Decoder** Turbo Code.
- **Reed Solomon Encoder/Decoder** Reed-Solomon Encoder/Decoder Block Code.
- **Convolutional Encoder** Convolutional Code.
- **Viterbi Decoder** Viterbi Code.

Waveform Applications need not provide these functions as part of the Waveform Application.

4.3.1 Turbo Encoder/Decoder

Figure 9 describes the interfaces for Turbo Encoder/Decoder. Implementation of these interfaces will be realized differently depending upon whether the processing element is a DSP or FPGA.

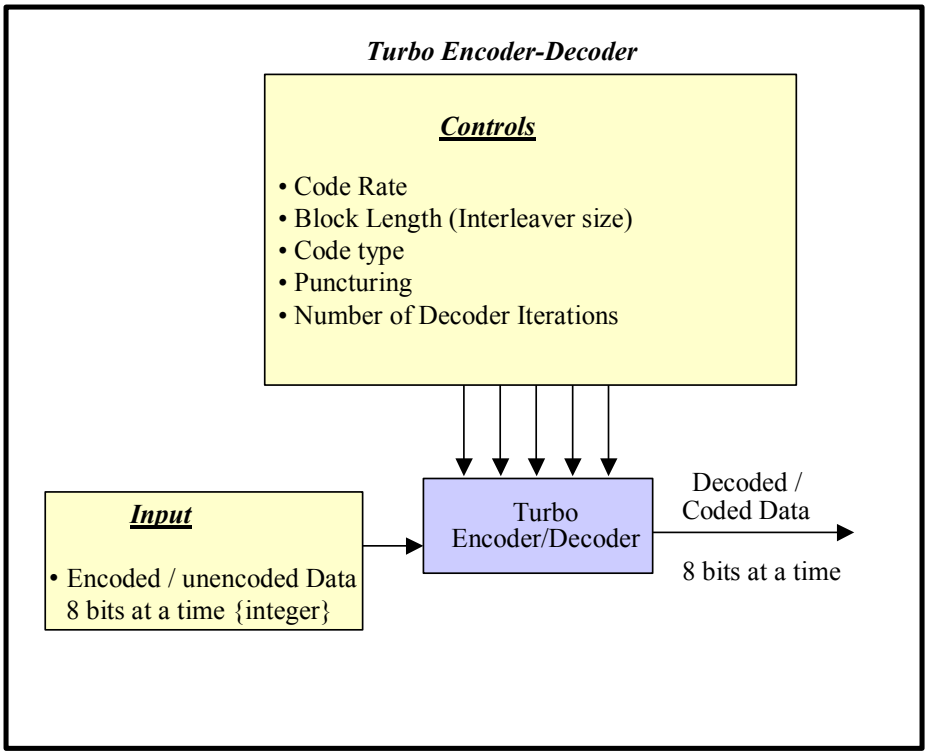


Figure 9: Turbo Encoder/Decoder

4.3.2 Reed-Solomon Encoder-Decoder

Figure 10 describes the interfaces for Reed-Solomon Encoder-Decoder. Implementation of these interfaces will be realized differently depending upon whether the processing element is a DSP or FPGA.

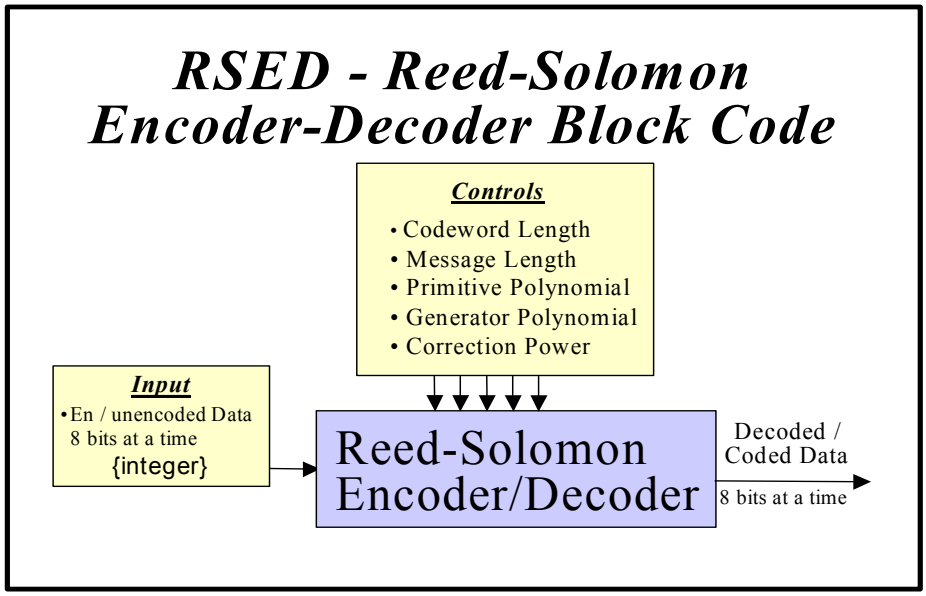


Figure 10: Reed-Solomon Encoder-Decoder

4.3.3 Convolutional Encoder

Figure 11 describes the interfaces for Convolutional Encoder. Implementation of these interfaces will be realized differently depending upon whether the processing element is a DSP or FPGA.

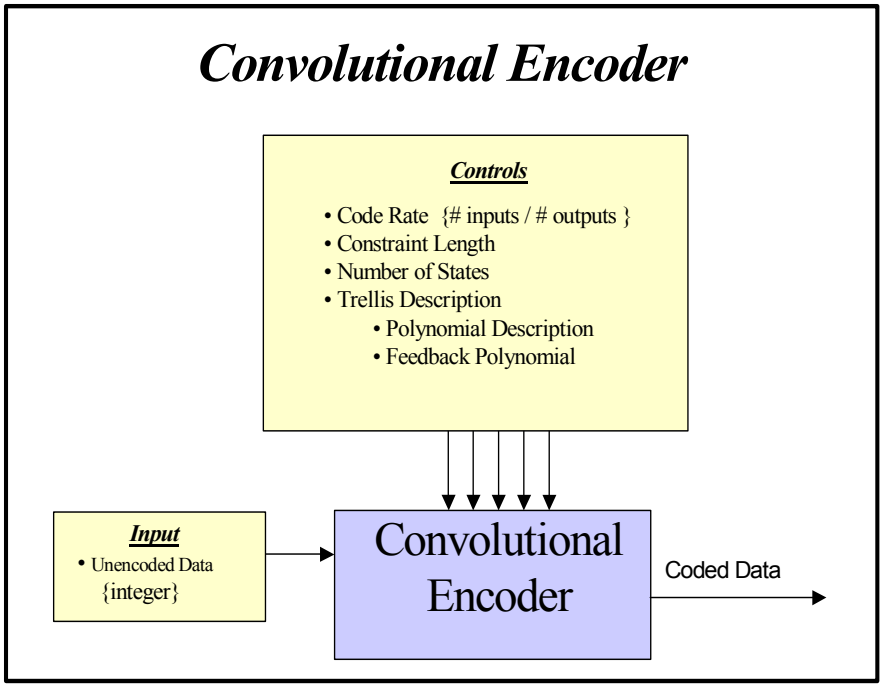


Figure 11: Convolutional Encoder

4.3.4 Viterbi Decoder

Figure 12 describes the interfaces for the Viterbi Decoder. Note: the Viterbi Decoder is usually used to decode Convolutional Encoded data. Implementation of these interfaces will be realized differently depending upon whether the processing element is a DSP or FPGA.

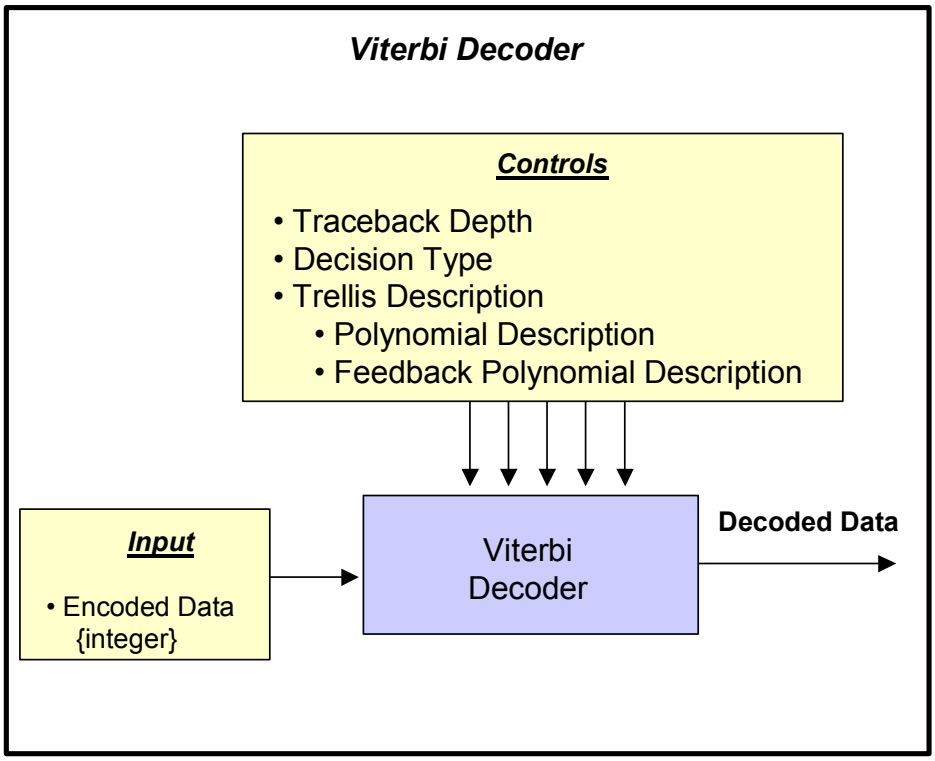


Figure 12: Viterbi Decoder

4.4 WAVEFORM FUNCTIONS FOR POTENTIAL STANDARDIZATION

The initial selection of Waveform Functional Blocks was limited to only those functions that were clearly used in several waveforms. The selection was conservative in order to avoid unnecessarily burdening JTR set developers in implementation of operating environments for DSP and FPGA processing elements.

Implementation of this initial set of common WFBs will provide a cost and benefit basis for determining whether additional WFBs should be standardized. The functionality found in existing waveforms (see Table 4 and Table 5) was the basis for the initial set of common WFBs and could provide additional candidate WFBs for future standardization.

Table 4: Standard Modulation Blocks

Modulation	FPGA	DSP
CCSK		WNW LPI
	LINK-16	
CPM		UHF SATCOM 181
		WNW BEAM
PSK		UHF SATCOM 181
BPSK		UHF SATCOM 183
OQPSK		UHF SATCOM 183
DQPSK		WNW OFDM
DOQPSK		WNW AJ
SOQPSK		UHF SATCOM 182 & 183
D16PSK		WNW OFDM
FSK	SINCGARS (CPFSK) All but Analog Voice	UHF SATCOM 181
FM	SINCGARS SCPT Analog Voice	
ASK		HAVEQUICK

Table 5: Standard Coding Blocks

	FPGA	DSP
Turbo Code	WNW BEAM	
		WNW OFDM
RSED		UHF SATCOM 181
		WNW OFDM
		WNW AJ
		WNW LPI
	LINK-16 RS(32,15) RS(16,7)	
Viterbi		UHF SATCOM 181, 182, & 183
		WNW AJ

Additional waveform functional blocks that are candidates for standardization include:

PSK Phase Shift Keying. For Non-DAMA UHF SATCOM 181, includes all BPSK and QPSK Modulations.

- **BPSK** Binary Phase Shift Keying.
- **OQPSK** Offset Quadrature Phase Shift Keying.
- **DQPSK** Differentially Encoded QPSK.
- **DOQPSK** Differentially Encoded Offset QPSK.
- **SOQPSK** Shaped Offset QPSK.
- **D16PSK** Differentially Encoded 16-ary Phase Shift Keying.
- **FSK** Frequency Shift Keying.

FM Frequency Modulation.

ASK Amplitude Shift Keying.

5 ANTENNA SUBSYSTEM API

5.1 PURPOSE AND SCOPE

This specification defines the Antenna Subsystem API that supports antenna control and status reporting. This API accommodates the characteristics associated with various antenna types such as directional, phased array, and smart antennas. These antennas are used for radio systems that include airborne, maritime, ground-mobile, and ground-fixed applications, as well as radios capable of 2 GHz and above.

Extension of this antenna API will foster software portability, deployment, reuse of antenna subsystems, and assist in allowing modular upgrades of antenna subsystem technology. It also facilitates efficient partitioning of the antenna control function implementation among GPPs, the signal processing subsystem, and the antenna subsystem in radio systems that require real-time antenna control.

Several sections of the OMG Software Radio Submission dtc/04-05-04 formed a basis for developing real time antenna control/status reporting APIs. Material was included from the following dtc/04-05-04 sections:

- Communication Equipment (8.2) – Identifies antenna I/O.
- Antenna (8.2.6.4.1) – Identifies antenna I/O device.
- Physical Layer Facilities (9.5.2) – Identifies physical layer functions.

5.2 ANTENNA API

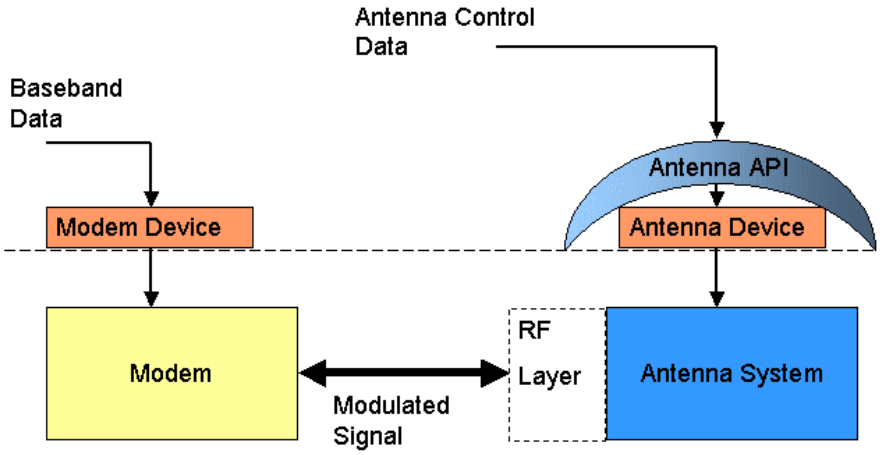
Antennas will require an interface to control and report different aspects of their design. Examples of real-time control and status reporting interface functions for “complex” antenna subsystems that currently exist in radio systems include:

- Antenna pointing commands to the antenna subsystem. Variants in this category include antenna-positioning commands (each axis) or directional pointing command of a phased array antenna embedded beam steering processor. Allowances for timing latency and jitter are also included in the interface requirements and are dependant on the dynamics of the radio set hardware and the link far-end radio set.
- Antenna subsystem status data necessary for antenna control. The data returned is dependent on the antenna type. Examples include rate sensor outputs and servo position feedback.
- Antenna subsystem status data from “smart” antennas, e.g., those that employ adaptive nulling processing in the antenna subsystem.
- Provisions for simultaneous operations with multiple antennas and multiple waveforms.
- Provisions for dynamic switching of the active transmit and/or receive antenna while maintaining communications.

These functions are more common in radios that have frequency response above 2 GHz and require antennas with directivity.

5.2.1 Antenna Control API

As shown in **Figure 13** the Antenna API is positioned between the Antenna device and waveform or radio system services communicating with the Antenna. A common API is provided to these waveforms and services. The device transforms the data to the formats expected by the antenna and communicates with the antenna via a device driver. **Figure 13** also shows that some control information is passed through the modem synchronized with the data transfer (see section 5.2.3).



NOTES: The Antenna Device will hide the hardware details that make up the Antenna system. Commands will be sent via the Antenna API to control the Antenna.

Figure 13: Positioning of Antenna API

The Antenna Control API depicted in **Figure 14** and described in this section has been registered as a JTRS API. Future JTRS acquisitions are required to use this API and to submit Change Proposals for any necessary corrections or extensions.

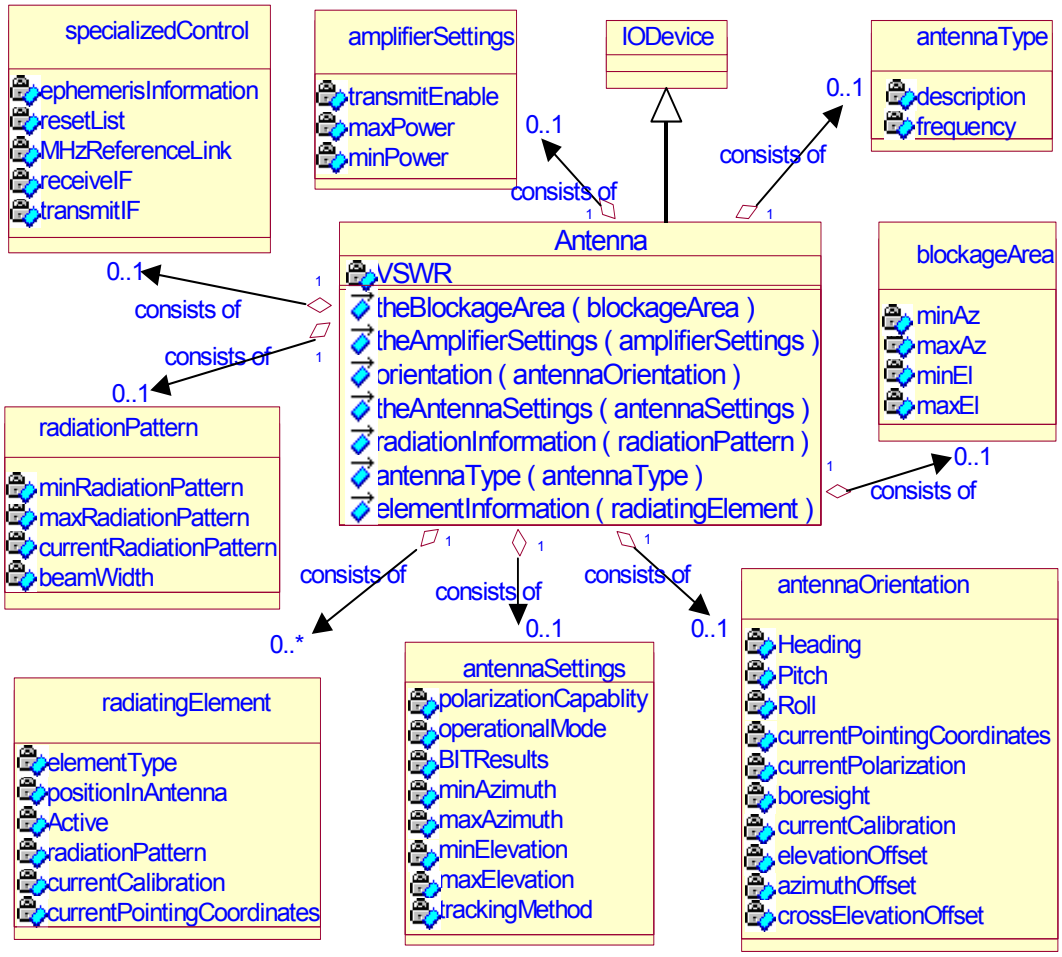


Figure 14: Antenna Command and Control UML diagram

blockageArea defines the null area of the antenna. There may be zero or one area with the following values:

minAz	float, minimum azimuth in radians
maxAz	float, maximum azimuth in radians
minEl	float, minimum elevation in radians
maxEl	float, minimum elevation in radians

amplifierSettings define the transmission power and mode of the antenna. An antenna may recognize at most one power range with the following values:

maxPower	float, maximum power in decibels (dBm)
minPower	float, minimum power in decibels (dBm)

The mode of the antenna will be defined as follow:

transmitEnable	enumeration (notEnabled, receive, transmit, both)

antennaOrientation defines the all aspects of the "pointing" of the antenna. There may be zero or one orientation with the following values:

heading	integer, direction of platform in degrees
pitch	integer, in degrees, the angle bow above stern
roll	integer, in degrees, the angle right wing above left
currentPointingCoordinates	float, direction (xyz) antenna is pointing from platform in radians
currentPolarization	enumeration (horizontal, vertical, lefthand, righthand)
boresight	integer, pointing adjustment in degrees
currentCalibration	integer, pointing adjustment in degrees
elevationOffset	integer, pointing adjustment in degrees
azimuthOffset	integer, pointing adjustment in degrees
crossElevationOffset	integer, pointing adjustment in degrees

antennaSettings defines the antenna capabilities and an overall indication of BIT results. There may be zero or one antennaSettings with the following values:

polarizationCapability	boolean, yes / no
operationalMode	enumeration
BITResults	boolean, pass /fail
minAzimuth	float, minimum azimuth in radians
maxAzimuth	float, maximum azimuth in radians
minElevation	float, minimum elevation in radians
maxElevation	float, minimum elevation in radians
trackingMethod	enumeration

radiationPattern defines the range of radiating patterns and the 3 dB beam width. There may be zero or one radiationPatterns with the following values:

minRadiationPattern	enumeration
maxRadiationPattern	enumeration
currentRadiationPattern	enumeration
beamWidth	float, 3 dB beam width in radians

antennaType provides a description of an antenna and gives the minimum and maximum frequencies that the antenna supports with the following values:

description	string
frequency	float, minimum and maximum in Hz

radiatingElement defines the aspects of a radiating element of a phased array. There may be zero or many radiatingElements with the following values:

elementType	enumeration (helix, flatpannel…..)
positionInAntenna	integer, position (xyz) coordinates in 100ths of meters
active	boolean, on/off
radiationPattern	radiationPatternType
currentCalibration	float, pointing adjustment in radians
	currentPointingCoordinates float, direction (xyz) antenna is pointing from platform in radians

specializedControl defines additional controls related to satellite communications. There may be zero or many Specialized Control with the following values:

ephemeris information	octets, 2 words specifying satellite position
resetList	array of enumerations indicating components to be reset
MHzReferenceLink	reference signal power level
receiveIF	receive IF signal power level
transmitIF	transmit IF signal power level

Implementers building the antenna constructs would implement the pieces needed to fulfill the specific radio set and waveform requirements. API aggregate class-types not needed for the radio set or waveform do not have to be instantiated.

5.2.2 SCA *Device* Support for Antenna Control API

There are SCA Device operations that support common *Device* functionality and would be used as part of an Antenna API. Extractions from the SCA (these are not new requirements) describing these operations and their use follow:

> **"Stop** - The *stop* operation is provided to command this Antenna to stop internal processing. The *stop* operation shall disable all current operations and put the Antenna in a non-operating condition. Subsequent *configure*, *query*, and *start* operations are not inhibited by the *stop* operation.
>
> **Start** - The *start* operation is provided to command this Antenna to start internal processing. The *start* operation puts the *Resource* in an operating condition.
>
> **RunTest** - The *runTest* operation initiates the tests in the Antenna subsystem. This allows Built-In Test (BIT) to be implemented and provides a means to isolate faults (both software and hardware) within the system."
>
> Synopsis.
>
> void runTest(in unsigned long testId, inout Properties testValues)raises (UnknownTest, UnknownProperties);

The *runTest* operation shall use the testId parameter to determine which of its predefined test implementations should be performed. The testValues parameter CF Properties (id/value pair(s)) shall be used to provide additional information to the implementation-specific test to be run. The *runTest* operation shall return the result(s) of the test in the testValues parameter."

5.2.3 Real-Time Antenna Control API

In addition to the antenna control API, some control information can be provided in synchronization with the data. This control information will need to be sent with the data since the user API might not be fast enough. This information is typically sent to the antenna directly from the modem so that it can be properly integrated with data transmittal or reception. Examples of Real-Time Control data for antennas are provided for information:

UpLink

antennuatorAdjustment	integer, number of dB adjustment
highFrequencyInhibit	boolean, normal/inhibit
pingPongSwitch	boolean, noswitch/switch
hopControl	integer, rate and frequency

DownLink

antennuatorAdjustment	integer, number of dB adjustment
switchBandwithFilter	enumeration of selectable filters
pingPongSwitch	boolean, noswitch/switch
hopControl	integer, rate and frequency
receiveSignalStrength	integer, to control antenna gain

APPENDIX A: HAL-C API

```
enum halcResult   {halcSUCCESS = 0,
                   halcFAIL,
                   halcTIMEOUT,
                   halcOVERFLOW};

halcResult
  halcGetEndpoint(
            halcEndpointHandle* eph, char*  componentName,
            char*  endpointName);

halcResult
  halcSend(
           halcEndpointHandle eph, uint32  maxLength,
           uint32* bytesSent, void*   data, uint32  timeout);

halcResult
  halcReceive(
              halcEndpointHandle eph, uint32 maxLength,
              uint32*  bytesReceived, void* data, uint32 timeout);


typedef
void (*halcCallback)(
              void* token,
              halcResult  status,
              uint32      length,
              void*       data);

halcResult
  halcRegisterCallback
              (halcEndpointHandle eph, halcCallback  cb,
               void*  token, uint32 timeout);
```